IFT 1010 - Programmation 1

Algo 1

Sébastien Roy & Balázs Kégl

Département d'informatique et de recherche opérationelle
Université de Montréal
automne 2003

Au programme

[Tasso :7] et [Niño : 2.3-6]

- Algorithme sur les tableaux
- Recherche
- Tri
- Complexité

Algorithmes & tableaux

On peut copier, déplacer, etc. les éléments d'un tableau.

Les deux applications les plus importantes des tableaux :

Recherche
: Trouver où un élément se trouve dans un tableau

Tri Placer les éléments d'un tableau selon un ordre donné

Algorithmes de recherche

Il existe de nombreux algorithmes de recherche.

Trois classes distinctes

- Recherche dans un tableau **non trié**
  (on ne connait rien sur l'ordre des éléments)
- Recherche dans un tableau **trié**
  (on sait que les éléments sont ordonnés)
- Recherche dans un tableau **semi-trié** (? hummm....)
  (on connait certaines informations sur les éléments)

Le Code

```
// on cherche la valeur v dans le tableau t
// on retourne l'index de v trouvé dans le tableau
// on retourne -1 si pas trouvé
int Recherche ( int t[], int v ) {
      ...
}
```

Les algorithmes de recherche (présentés ici)

Tableau non-trié

- recherche directe

Tableau trié

- recherche linéaire
- recherche binaire

Tableau semi-trié

(dépend des informations connues sur le tableau)

Algorithmes de recherche directe

Idée : passer séquentiellement tous les éléments

```
int RechercheDirecte ( int t[], int v ) {
    int i;
    for( i=0 ; i<t.length ; i++ ) {
        if( t[i] == v ) return i;  // trouvé! :-)
    }
    return -1; // pas trouvé! :-(
}
```

Voici la même version (avec bug inclus !) :

```
int RechercheDirecte ( int t[], int v ) {
    int i=0;
    while( ( t[i] != v ) && ( i < t.length ) ) i++;
    if( t[i]==v ) return i; // trouvé!
    else return -1; // pas trouvé!
}
```

Algorithmes de recherche directe

Un exemple tiré d'un autre cours de Java....
Que dites-vous de la lisibilité de ce code ?

```
int RechercheDirecte ( int t[], int v ) {
    final int TROUVE = 0, ABSENT = 1, CHERCHE = 2;
    int i = 0, etat = CHERCHE;

    do {
        if( i >= t.length ) etat = ABSENT;
        else if( t[i] == v ) etat = TROUVE;
        else i++;
    } while( etat == CHERCHE );
    switch( etat ) {
        case TROUVE: return i;
        case ABSENT: return -1;
    }
}
```

Performance de la recherche directe

Mesure : On compte le nombre de comparaisons effectuées

Pour un problème de taille N (nombre d'éléments) :

Meilleur cas

On tombe sur le bon élément du premier coup : 1 opération.

Pire cas

On passe tout le tableau : N opérations.

Cas moyen

La valeur peut être n'importe où avec probabilité uniforme : $N/2$ opérations.

Algorithmes de recherche linéaire

Idée : on peut arrêter de chercher avant la fin
(on suppose que les éléments sont en ordre croissant)

```
int RechercheLinéaire ( int t[], int v ) {
    int i;
    for( i=0 ; i<t.length ; i++ ) {
        if( t[i] == v ) return i;  // trouvé!     :-)
        if( t[i] > v ) return -1;  // pas trouvé! :-(
    }
    return -1; // pas trouvé! :-(
}
```

Si l'élément est absent du tableau, on arrête plus rapidement que la recherche directe.

Algorithmes de recherche binaire

Idée : On va sauter un peu partout dans le tableau
(un peu comme chercher dans le dictionnaire...)

- On choisi une page au milieu du dictionnaire
- On regarde si le mot cherché est avant ou après cette page
- On cherche maintenant dans la une moitié du dictionnaire

Plus précisement :

- Choisir un intervalle de pages couvrant le dictionnaire $[a, b]$
- Visiter la page milieu de cet intervalle, $c = (a + b)/2$.
- Si le mot est trop petit, on reprend dans l'intervalle $[a, c]$
- Si le mot est trop grand, on reprend dans l'intervalle $[c, b]$

Algorithmes de recherche binaire

```
int RechercheBinaire ( int t[], int v ) {
    int a=0;            // premier élément
    int b=t.length-1; // dernier élément
    int c;            // point milieu

    while( a <= b ) {
        c=(a+b)/2; // milieu!
        if( t[c] == v ) return(c); // trouvé! :-)
        if( v < t[c] ) b=c-1; // v est trop petit! -> [a,c-1]
        else           a=c+1; // v est trop grand! -> [c+1,b]
    }
    return -1;  // pas trouvé! :-(
}
```

Pourquoi le −1 et +1 ?

Performance de la recherche binaire

Mesure : On compte le nombre de comparaisons effectuées

Pour un problème de taille N (nombre d'éléments) :

Meilleur cas

On tombe sur le bon élément du premier coup : 1 opération.

Pire cas

On divise N par deux jusqu'à ce qu'il soit égal à 1 : $\log_2 N$ opérations. Ex : Si $N = 32$, alors on peut diviser 5 fois par 2 (donc par 2^5) pour obtenir 1.

Cas moyen

Hummmm.... On remet ça à plus tard....
(mesurez expérimentalement, c'est facile !)

Une amélioration de la recherche binaire...

Il serait utile de savoir où on doit insérer un élément dans le tableau...

$\Rightarrow$ Modifier la recherche binaire pour qu'elle retourne $(-i-1)$ si la valeur v n'est pas dans le tableau et devrait être ajoutée à la position i.

Algorithmes de tri

Pour chercher efficacement, il faut commencer par trier les éléments.

- Tri par sélection
- Tri par insertion

Idée commune :

- On sépare le tableau en deux partie : une triée, une non triée.
- Au départ la partie triée est vide, la non triée fait tout le tableau.
- On transfert un par un les éléments de la partie non triée vers la partie triée.

Algorithmes par sélection

Idée : On sélectionne l'élément non trié minimum et on le place à la fin de la partie triée

```
void TriSelection( int [] t ) {
    int i; // position pour placer le minimum
    int j; // index pour trouver le minimum
    int jmin; // position trouvee du minimum

    for( i=0 ; i<t.length-1 ; i++) {
        // les éléments [0..i-1] sont déjà triés
        // choisi le plus petit élément du tableau [i..N-1]
        jmin=i;
        for( j=i+1 ; j<t.length ; j++) {
            if( t[j] < t[jmin] ) jmin = j;
        }
        // t[jmin] est le minimum, on échange avec t[i]
        if( jmin != i ) { z=t[jmin]; t[jmin]=t[i]; t[i]=z; }
    }
}
```

Algorithmes par insertion

Idée : On insère un à un les éléments non triés au bon endroit dans la partie triée du tableau.

```
void TriInsertion( int [] t ) {
    for( i=1; i<t.length; i++) {
        // on sait que le tableau [0..i-1] est déjà trié
        // on veut insérer l'élément [i] dans le tableau [0..i-1]
        // c'est le plus petit j dans [0..i-1] tel que t[j]>t[i]
        // = 1+le plus grand j dans [0..i-1] tel que t[j]<=t[i]
        for( j=i; j>=1 ;j--) {
            if( t[j-1]<=t[i] ) break;
        }
        if( j==i ) continue; // rien a deplacer!
        z=t[i]; // sauvegarde la valeur de [i]
        // Faire un trou en [j] en déplacant [j..i-1] -> [j+1..i]
        for( k=i-1; k>=j; k-- ) t[k+1]=t[k];
        t[j]=z; // place la valeur de [i] dans le trou en [j]
    }
}
```

Sélection VS Insertion

Quelle est la performance ?

Le tri par sélection performe toujours le même nombre d'opérations :

$$
\begin{aligned}
\sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} 1 &= \sum_{i=0}^{N-2} (N-1-i) \\
&= (N-1)(N-1) + \sum_{i=0}^{N-2} (-i) \\
&= (N-1)(N-1) - (N-2)(N-1)/2 \\
&= (N-1)((2N-2) - (N-2))/2 \\
&= (N-1)(N)/2
\end{aligned}
$$

Performance du tri par insertion

Meilleur cas

Si le tableau est déjà trié, on visite chaque élément une seule fois : $\approx N$ opérations

Pire cas

Si le tableau est déjà trié en ordre décroissant : $\approx N^2$ opérations.

Cas moyen

Hummmm.... On remet ça à plus tard....
(mesurez expérimentalement, c'est facile !)

Mais il existe d'autres algorithmes de tri avec $\approx N \log N$ opérations...

Choix de l'algorithme

Il est important de bien choisir un algorithme.

Soit `a` et `b` deux tableaux non-triés.
Trouvez le maximum de la différence absolue entre deux éléments de ces tableaux.
$\Rightarrow$ Trouvez $\max_{i,j}(|a[i] - b[i]|)$.

Solution évidente :

```
max=-1;
for( i=0; i<a.length; i++ ) {
    for( j=0; j<b.length; j++ ) {
        k=Math.abs(a[i]-b[j]);
        if( k > max ) max=k;
    }
}
```

Prend $\approx N^2$ opérations.

Choix de l'algorithme

Il existe un meilleur algorithme...

```
amin=amax=a[0];
for( i=1; i<a.length; i++ ) {
    if( a[i] < amin ) amin=a[i];
    if( a[i] > amax ) amax=a[i];
}
bmin=bmax=b[0];
for( i=1; i<b.length; i++ ) {
    if( b[i] < bmin ) bmin=b[i];
    if( b[i] > bmax ) bmax=b[i];
}
if( (amax - bmin) > (bmax - amin) ) return (amax - bmin);
else return (bmax - bmin);
```

On trouve les minimum et maximum de chaque tableau séparement ! $\Rightarrow$ Prend $\approx N$ opérations.

Et si les tableaux étaient triés ? $\Rightarrow$ Prend ≈ 1 opération